A Formal Theory for the *Views-a* Relationship

P.S.C. Alencar D.D. Cowan L.C.M. Nova
{alencar,dcowan,luisnova}@csg.uwaterloo.ca
Computer Science Department
University of Waterloo
Waterloo, ON, N2L3G1, Canada

C.J.P. Lucena
lucena@inf.puc-rio.br
Departamento de Informática
Pontifícia Universidade Católica
Rio de Janeiro, RJ, 22453-900, Brazil

Abstract

The lack of methodological support for reuse has been identified as one of the major causes why software developers can not take full advantage of reuse pay offs such as software productivity, quality, and cost improvement. There is, therefore, a need for explicit definitions about how to practice reuse as part of the development process. These definitions include models and properties of reuse techniques which can improve understanding and provide more guidance for the software developers that want to adopt reuse. In this paper we provide a model and properties related to a reuse technique called separation of concerns that can be applied in object-oriented design. The formal model defines how to combine classes or components dealing with different concerns in object-oriented design so that separation of concerns is achieved. The model is presented as a design relationship which is characterized by a set of properties. These properties can be seen as a semantics for gluing object-oriented components with separation of concerns in mind. We illustrate the model by presenting some of the typical properties that describe the combination. We also show how the model can be used in user interface design.

1 Introduction

There is a wide consensus that software reuse enables significant software productivity, quality, and cost improvement [14, 15, 17]. However, most software methodologies do not include support for reuse [15]. Thus, there is a need for explicit definitions about how to practice reuse as part of the development process. These definitions include models and properties of reuse mechanisms that can clarify and provide guidance for the software developers that want to adopt reuse.

In this paper we deal with a reuse technique called separation of concerns [1] that can be applied in object-oriented design. Separation of concerns is a well-established principle in software engineering that attempts to hide complexity through abstractions [16] that carefully segregate different aspects of a set of related algorithms such as those encompassed in an object.

Current software applications can be separated into one or more basic concerns and a number of special purpose concerns. The fundamental computational algorithms that provide the essential functionality relevant to an application domain represent the basic concerns. The special purpose concerns relate to other software issues, such as user interface presentation, control, timing, and distribution. Special purpose concerns are extensions to the basic functionality that fulfill special requirements of the application, or enhance, manage, restrict, or optimize the basic algorithm [12].

By segregating these different aspects of an object we create objects that do not embed any knowledge about concerns such as being shared or being presented through a user interface. Thus, by applying the principles of separation of concerns we localize different kinds of information in the software descriptions making them easier to write, understand, reuse, and modify.

Object-oriented programming languages do not naturally support separation of concerns. We need to create design abstractions that address separation of concerns where these constructs can be mapped into a programming paradigm at implementation time. In this way, separation can be captured and the objects at the design level of specifications can be reused.

Separation of concerns and design models that support separation of concerns have been discussed in an informal setting, in which concepts, definitions, and guidelines have been provided. An informal discussion about the problem of separation of concerns in object-oriented modeling is provided in [12]. However, a formal approach to the characterization, definition, use, and combination of concerns has not been addressed.

“Views” and the related operation “views-a” are formal design constructs that support the designer in segregating the basic and special concerns. The basic concerns are encapsulated in the object, while the special concerns are placed in views that are related to the object through the views-a operation. Views are objects with extra properties that act as interfaces to the external world such as user interfaces, or as interfaces between objects [3].

Views-a can be seen as a constrained association between an object and one of its views where the association has a well-defined set of semantics. With an object calculus relating objects and views [3] and the accompanying semantics, we can validate our design. The semantics also provide a way of mapping the view and *views-a* design constructs into an equivalent implementation using objects. This mapping is normally achieved thorough the use of design patterns [11].

Specialization through inheritance is an approach that is suggested to achieve separation of concerns. However, inheritance forces all the concerns to be embedded somewhere in the class structure, and unless very carefully used, does not allow the object to be distinguished from all of its related special conditions. In contrast views and *views-a* were conceived specifically to address issues related to separation of concerns.

In our approach to developing object-oriented systems, the software designer defines how to separate concerns and encapsulate them in classes or object-oriented components. The view is used to capture the special concerns, and then the *views-a* operation acts as the “glue” to relate each view to the corresponding object that encapsulates the basic concern. The high-level design is then mapped into an object implementation through the application of design patterns. In this way the concerns and their relationship are clearly delineated.

Views and *views-a* were first addressed in 1992 when we formulated the concept of the abstract data object (ADO) and abstract data view (ADV) as an approach to encapsulating the basic and special concerns in an object-oriented software design. These concepts first appeared in [7, 8].

Figure 1: Simple clock models.

Because these were design rather than programming constructs we later changed the name to abstract design objects and abstract design views in a 1995 paper [9]. A graphical design approach similar to OMT [18] was proposed in several publications [6] in 1994 and 1995. Because of its explicit use in separating concerns we have also shown how the concepts can be applied to many other areas in software design including software maintenance, viewpoints and viewpoint analysis. A complete formal theory for object and interfaces using category theory and based on the concept of view and *views-a* is presented in [3].

In this paper we describe a formal model for views and *views-a* that addresses separation of concerns and defines how to combine or glue classes or components dealing with different concerns into object-oriented design so that separation of concerns is achieved. We illustrate the model by presenting some of the typical properties that describe the combination, and also show how the model can be used in a user interface design.

2 Motivation

Within current design models, a single type of modeling construct can be used for several different purposes. More specifically, relationships between classes and/or objects are applied without a clear presentation of their semantics. Unfortunately, these various purposes and semantics are not supported by current object-oriented methods, which often represent them under a single notational construct. This situation inevitably creates misconceptions in intent which often prove costly throughout the development process.

Specialization is a class relationship in which the behavior of a superclass is shared by all of its subclasses. In a proper specialization, an operation of the subclass that corresponds to an operation in the superclass is responsible for providing equivalent services, and extensions. However, specialization is used as a technique to reuse some behavior, even when the related classes are inherently different. This misuse of this relationship can lead to all sorts of problems, including ones related to understandability.

Let us present a simple example to clarify our point, while having in mind the situation described in the previous section that a designer faces when modeling with separation of concerns. We deal here with two concerns: the application (a counter) and the user interface (a clock view). Figure 1 shows two models of a clock system which are based on a *Clock View* object using different kinds of object-oriented relationships to access attributes and operations in a *Counter* object. The first model, illustrated by Figure 1(a), is an example of improper use of specialization. In this case, a *Clock View* class inherits both desirable and undesirable operations from a *Counter* object. The

Figure 2: Yet another clock model.

desirable operations are used, while the undesirable ones are masked or just ignored. This may lead to serious misconceptions and possible unexpected behavior.

An alternative design is shown in Figure 1(b). In this model, the two classes involved will be instantiated as distinct objects, with *Clock View* delegating its behavior to the appropriate operations of *Counter*. This model guarantees that an *adjust* operation requested to *Clock View* will be delegated to *Counter* and properly executed. Undesirable operations of *Counter* will not be accessible through the *Clock View* interface.

For a number of reasons, the two models in Figure 1 represent inappropriate solutions. In terms of expressiveness, an interpretation of the model based on an aggregation relationship indicates that a clock view object *is composed of* a counter. Alternatively, the model based on the specialization relationship indicates that a clock view *is a kind of* counter. Both interpretations are inaccurate.

In addition, the properties of the relationships used in those previous models introduce a variety of design constraints to the specification of an interface and the application it represents. For instance, in both models only a single clock view may exist for each instance of a counter. This means that the system developer will not be able to add a different perspective or view to the *Counter* object, as, in both models, the *Clock View* and *Counter* are tightly coupled.

An alternative solution is provided by the *views-a* approach which separates objects with different concerns. In such approach, a *views-a* relationship represents a particular type of object interconnection that emphasizes and rules the distinction between concerns. Figure 2 shows the *views-a* relationship providing an alternative solution to the *Clock View* modeling problem also illustrated in Figure 1. The specific meaning of the *views-a* relationship indicates that the *Clock View* object is providing a different perspective (*view*) for the use of a *Counter* object. Rather than implying strong relations like *is composed of* or *is a kind of*, the mechanism characterizes *Clock View* as a particular point of view for a *Counter*. Some typical properties of such a relationship include multiplicity of views, consistency between the state of the view and the state of the application, restrictions on the creation and deletion of views, and visibility between view and application. In later sections we formally describe a set of properties characterizing such relationship.

3 The Categorical Framework

In this section we describe the theoretical background that is used to provide a formal interpretation theory for the *views-a* relationship. This background consists of a categorical framework together with object calculus theories based on logic proposed in [5, 10]. We will use the categorical framework to illustrate how the characteristics of separation of concerns can be described by a formal relationship and how these concerns can be combined within a logic-based formalism. The

object calculus theories model the components of the system in terms of signatures and logic axioms. This framework was chosen as the formal underlying description because it allows us to isolate the properties of the relationship we want to model. In what follows, for completeness, we briefly describe the aspects of the categorical framework needed to understand this paper.

An object description is defined as an interpretation theory in temporal logic: a signature θ and a set of axioms Φ . The theory of a class is given by a combination of two distinct theories: class instances and class manager. A typical class instance theory represents the theory for every object of this class. This theory introduces sorts for the type of each attribute ($\mathcal{S}$), the attributes ($\mathcal{A}$), and the actions ($\mathcal{G}$).

Note that one important constraint to be satisfied by each object description in the object-oriented approach is given by the locality axiom. This axiom guarantees the encapsulation of attributes in an object. In other words, the attributes (state) of an action may be modified only by actions which are local to the object. Such locality requirement is specified by the following axiom. For every signature $\theta = \langle \mathcal{S}, \mathcal{A}, \mathcal{G} \rangle$:

$$Locus_\theta : ((\bigvee_{g \in \mathcal{G}} (\exists x_g) g(x_g)) \vee (\bigwedge_{a \in \mathcal{A}} (\forall x_a) (\bigcirc a(x_a) = a(x_a)))),$$

where for each symbol u , x_u is a tuple of distinct variables of the appropriate sorts.

The second theory for the specification of a class is called class manager. A class manager theory M controls the creation and destruction of instances of a class. For a general class type X , the theory introduces a sort for identifiers of objects called $@X$. The $@X$ sort is a set of identifiers for any possible instance of X , which includes currently existing and non-existing instances of X . The set of currently existing instances is defined by an attribute $\overline{X}$ of M . The class manager theory also specifies actions to create and kill objects of X . The following is the signature for M :

$$\begin{aligned} \mathcal{S} &= \{@X\} \\ \mathcal{A} &= \{\overline{X} : \mathbb{F}@X\} \\ \mathcal{G} &= \{create : @X, kill : @X\} \end{aligned}$$

The pre- and post-conditions to the actions in theory M are summarized in the following temporal logic axioms:

$$create(x) \Leftrightarrow x \notin \overline{X} \wedge x \in \bigcirc \overline{X} \quad (1)$$

$$kill(x) \Leftrightarrow x \in \overline{X} \wedge x \notin \bigcirc \overline{X} \quad (2)$$

In addition to these axioms, an initialization rule states that the initial set of existing instances of X is empty. This is formalized by:

$$\mathbf{BEG} \Rightarrow \overline{X} = \emptyset \quad (3)$$

Following the formal object theory in [5], we use a morphism to combine the theory of each instance with the theory of the class, as illustrated in Figure 3. As a result, a **self** identifier – which is the name an object refers to itself – is mapped by the morphisms to global identifiers, such as x_i , inside the class theory. In addition, each attribute and action symbol of an instance will have an extra parameter for identification. For example, an attribute *attr* of a class instance x_i is conveniently identified as $x_i.attr$. A rigorous specification for the combination of every two object theories is shown in the next section.

Figure 3: Morphisms between instances and class theories.

3.1 Combining Object Theories through Morphisms

As previously described, a category theory describing the whole system is composed of object and morphism theories. While in the previous section we described some temporal logic axioms for the specification of objects, we now concentrate on the concepts of morphisms combining objects in a category.

A morphism between theory presentations (or a description morphism) is a signature morphism that defines a theorem-preserving translation between the two theory presentations, and also preserves the translation of the locality axiom. These morphisms can be used to express a system as an interconnection of its parts, that is, as a diagram. This diagram is a directed multigraph in which the nodes are labeled by theory specifications, and the edges by the specification morphisms. Figure 3 illustrates an initial diagram that shows the morphisms between each object instance A_i of a class theory A and the class theory A itself. This diagram will be later complemented with the addition of other theories, thus composing a complete system.

We can reduce a diagram of specifications to a single specification by taking the colimit of a diagram. Informally, the colimit of a diagram is the disjoint union of all specifications (attributes, actions and axioms), together with the identification of some attributes and action symbols that receive the same name. For example, if two attributes $attr_A$ and $attr_B$, have been identified they receive the same name $attr$ in the resulting colimit. Technically, the colimit of a diagram is constructed by first taking the disjoint union (coproduct) of all the specifications in the diagram and then the quotient of this coproduct via the equivalence relation generated by the morphisms in the diagram. The result is a valid specification because the colimit exists.

We now show how combinations of object theories can be achieved with categories. As previously described, morphisms are used to express relationships between the component objects, and the composite object is obtained by constructing the colimit of the diagram that represents the interaction among the objects. We exemplify the construction of the colimit in the case we have two object theories A and B . These two theories interact through a common subcomponent S which synchronizes the interaction. Synchronization in this case identifies actions in A with actions in B . We create an object theory S and two morphisms f and g such that $f : S \rightleftarrows A$ and $g : S \rightleftarrows B$. This combination is represented by the following diagram:

$$A \rightleftarrows^f S \rightleftarrows^g B.$$

In order to obtain the object describing the combination of A and B , we build the colimit of

Figure 4: Morphisms forming a composite of two object theories.

this diagram. In this particular case, as we are only dealing with two elements, the colimit is called a pushout. Pushouts are just an example of a combination of object theories by assembling them in a diagram and connecting them through the appropriate interfaces.

Figure 4 illustrates a complete diagram of the combination of two object theories to generate a composite object C.

3.2 An Interpretation Theory for a Relationship

Having defined an interpretation theory for a class that combines class instances and class management theories, in this section we describe an object calculus theory that relates objects defined by these class theories. Even though, the object relationship theory uses the identities of the related objects, the interpretation theory for this general relationship is independent of the structure of the objects it relates. For more complex relationships, such as the *views-a* relationship, there will be a need for additional properties involving elements of the object structure.

The theory is based on a relationship between objects of classes R and D . The identifiers of objects in these classes are, respectively, in $@R$ and $@D$. A signature for the general relationship theory follows.

$$\mathcal{S} = \{@R, @D\}$$

$$\mathcal{A} = \{rd : \mathbb{F}(@R \times @D)\}$$

$$\mathcal{G} = \{link : @R \times @D, unlink : @R \times @D\}$$

Note that rd is an attribute that represents the set of currently existing relationships, and $link$ and $unlink$ are the only actions capable of creating or removing a relationship. Similarly to the axioms for the creation and deletion of objects, the pre- and post-conditions for $link$ and $unlink$ are:

$$link(r, d) \Leftrightarrow (r, d) \notin rd \wedge (r, d) \in \bigcirc rd \quad (4)$$

$$unlink(r, d) \Leftrightarrow (r, d) \in rd \wedge (r, d) \notin \bigcirc rd \quad (5)$$

$$\mathbf{BEG} \Rightarrow rd = \emptyset \quad (6)$$

The previous axioms do not assume any condition on the state of the related objects. However, one condition for the existence of a relationship is that the objects involved are currently alive. This is stated by:

$$\forall r \in @R, d \in @D \cdot (r, d) \in rd \Rightarrow r \in \overline{R} \wedge d \in \overline{D} \quad (7)$$

Figure 5: Diagram showing the colimit of object and relationship theories.

which, together with axioms 4 and 5, yields the following post- and pre-conditions for the relationship theory actions:

$$link(r, d) \Rightarrow r \in \bigcirc \overline{R} \wedge d \in \bigcirc \overline{D} \quad (8)$$

$$unlink(r, d) \Rightarrow r \in \overline{R} \wedge d \in \overline{D} \quad (9)$$

Combining object and relationship theories. In the previous section we described the formal approach to the combination of object theories. In Figure 4 we showed a simple example where two object theories A and B are combined to form a composite object C, which contains the theory of both A and B. Now, some complexity is added to the subsystem as we introduce the theory of a relationship.

Figure 5 shows the diagram where two object class theories R and D and one relationship theory V are interconnected by morphisms to derive the composite theory C . C is interpreted as the colimit of such diagram. Note also that the theories responsible for synchronizing both class theories with the relationship are the class manager theories, which were described in Section 3. These class manager theories may be interpreted as the “glue” that combines objects and relationship theories.

We now investigate the set of properties which interpret the *views-a* relationship, which relates viewer ($@R$) and viewed ($@D$) objects.

4 Properties of the *Views-a* Relationship

Each different kind of object-oriented relationship “glues” objects in a peculiar way. The semantic properties of these relationships introduce static and dynamic constraints which characterize the type of interaction between the related objects. These constraints determine how one action occurring in the object in one end of the relationship affect the object in the other end. Our current interest is to specify the constraints of a *views-a* relationship V , where $V = \{\mathcal{S}_V, \mathcal{A}_V, \mathcal{G}_V\}^1$, between viewer and viewed objects.

¹ Note that the signature of V is the same as the general relationship described in the previous section.

4.1 Objects Identity

One simple rule for the *views-a* relationship is that viewer and viewed objects should have different identities. This implies that the viewer's actions and attributes cannot be mapped by morphisms to the inside of its own object structure. Thus, the *views-a* relation is irreflexive. Note, however, that the restriction is on the object identities, not on the object classes. This constraint is stated by:

$$link(r, d) \Rightarrow r \neq d \quad (10)$$

4.2 Cardinality Constraints

Cardinality is a constraint over the number of instances of a class during the execution of a system. As an example, for a typical association, the cardinality of the class at one end of the relationship may be set to any positive integer or be variable inside a non-negative range. It is a design decision made during system development.

While cardinality at both ends of an association may be designed in many possible ranges, other object relationships may establish more rigorous constraints. In the *views-a* relationship case, the viewed object d may be related to zero, one, or many viewer object during any time the system is being executed. However, when the viewer object is alive, i.e. $r \in \overline{R}$, it must be related to an object d . These constraints, in addition to the axioms 8 and 9, result in the following conditions on the actions of V :

$$link(r, d_1) \wedge link(r, d_2) \Rightarrow d_1 = d_2 \quad (11)$$

$$link(r, d) \Leftrightarrow r.create_R \quad (12)$$

$$unlink(r, d) \Leftrightarrow r.kill_R \quad (13)$$

From axioms 11 and 12 we also infer that V is a **function** from $\overline{R}$ to $\overline{D}$. With the addition of axiom 13, we may actually infer that *views-a* is a **total function** from $\overline{R}$ to $\overline{D}$.

From this statement, we identify one lifetime constraint between the related objects. This constraint states that an instance of a viewer class **R** will only exist if related to some object of a viewed class **D**, which is formally stated by:

$$\forall r \in \overline{R} \cdot (\exists d \in \overline{D} \cdot (r, d) \in rd)$$

4.3 Creation/Destruction of Objects and Relationship

In a system, objects do not exist in isolation. They interact and cooperate among themselves to accomplish a more complex task. In fact, there may be cases where one object is meaningless without the other. For instance, if an object is created to monitor changes in a given subsystem, this object will be unable to perform its task if the mentioned subsystem is not alive.

Views-a is a relationship that creates a unidirectional dependency between objects. Such a dependency is illustrated by the lifetime pre- and post-conditions for the related objects.

Viewer Lifetime Conditions. From the cardinality constraints previously specified, we can infer that the creation of a viewer object implies the creation of the *views-a* relationship in which this object participates, and vice-versa. Additionally, a lifetime constraint implies that if a viewer object is killed then the *views-a* relationship for this object is destroyed. The inverse is also true,

i.e., if a *views-a* relationship is destroyed, the viewer object in this relation is killed. These four conditions are stated in axioms 12 and 13.

While there is a strong correlation between the existence of a *views-a* relationship and its viewer elements, the same is not true for the viewed elements. In this regard, the only condition for the creation or destruction of a *views-a* relationship at a certain time is the existence of the viewed object at this time and, consequently, during the existence of the relationship. These conditions are defined in axioms 8 and 9.

Note that, as *views-a* and the viewer objects are strongly correlated, the conditions on the viewed object lifetime which result from the creation/destruction of a *views-a* relationship are also valid for the creation/destruction of a viewer object. Thus, for any $(r, d) \in V$, we have:

$$r.create_R \Rightarrow d \in \bigcirc \overline{D}$$

$$r.kill_R \Rightarrow d \in \overline{D}$$

Viewed Object Lifetime Conditions. As viewer objects may be modeled as optional (see Section 4.2), a viewed object may exist for a period of time without being related to any viewer. Therefore, the creation of a viewed object does not require any pre-condition related to the existence of the corresponding *views-a* relationship or the viewers.

On the other hand, the destruction of a viewed object implies the destruction of every related viewer object. For instance, suppose that a viewed object d , which is part of a *views-a* relationship (r, d) , is *killed*. The previous assertion is expressed as the following theorem:

Theorem 1 $\vdash \forall r \in \overline{R}, d \in \overline{D} \cdot (r, d) \in rd \wedge d.kill_D \Rightarrow r.kill_R$

Our proof starts with the inference from rule 2 that:

$$d.kill_D \Rightarrow d \notin \bigcirc \overline{D}$$

Using this rule and the contrapositive of axiom 7, we also infer that:

$$d.kill_D \Rightarrow (r, d) \notin \bigcirc rd$$

Combining this result and the hypothesis $(r, d) \in rd$, from axiom 5 we have:

$$unlink(r, d)$$

Axiom 13 implies:

$$r.kill_R$$

The above conditions are expressed in Table 1, which describes the lifetime-related conditions for the occurrence of each of the actions specified in the object and relationship theories.

		<i>pre-conditions</i>			<i>post-conditions</i>		
		<i>Views-a</i>	<i>Viewer</i>	<i>Viewed</i>	<i>Views-a</i>	<i>Viewer</i>	<i>Viewed</i>
<i>Views-a</i>	<i>created</i>	$\notin rd$	created	none	$\in rd$	none	exists
	<i>killed</i>	$\in rd$	killed	exists	$\notin rd$	none	none
<i>Viewer</i>	<i>created</i>	created	$\notin \bar{R}$	none	none	$\in \bar{R}$	exists
	<i>killed</i>	killed	$\in \bar{R}$	exists	none	$\notin \bar{R}$	none
<i>Viewed</i>	<i>created</i>	none	none	$\notin \bar{D}$	none	none	$\in \bar{D}$
	<i>killed</i>	killed	killed	$\in \bar{D}$	none	none	$\notin \bar{D}$

Table 1: Conditions on actions of objects and relationship

4.4 Viewed Singularity and Viewer Multiplicity

So far, we have analyzed the *views-a* relationship as an interconnection mechanism between two distinct classes. A complete system, however, is usually composed of several classes connected by several relationships of different types and semantics. In fact, it is usual that one single class is related to other classes in the system by means of a few different relationships. Sometimes, there are different occurrences of the same type of relationship. For instance, one class A may be related in an association to another class B. At the same time, class A may be associated with class C. In this case, we have two occurrences – namely *ab* and *ac* – of the same type of relationship (i.e., association).

While there is an unlimited number of associations for which a given class may take part, other relationships may have different constraints at one or both ends of the relationship. For instance, in many specification languages [13, 18], the definition of aggregation² implies that one object will not play the contained role for aggregation relationships more than once. In other words, an object will be contained in at most one container object. In a *vehicle* specification system, for example, a *wheel* object is contained in at most one *car* object, even though *wheel* may be the container of *tire* and *bolt* objects. The *views-a* relationship defines similar constraints on the objects being related.

As previously mentioned, it is a responsibility of the viewer to monitor the behavior of a viewed object according to rules defined by a *views-a* relationship. Such viewed behavior is characterized by changes occurring in its attributes values. Thus, the behavior monitoring performed by a viewer depends on the attribute values of the object being viewed. Such dependency on the viewed object structure creates constraints on the types of objects a viewer is capable of monitoring. An additional constraint on the *views-a* approach is that an object *r* will play the role of a viewer for at most one *views-a* relationship theory. This property is rigorously stated by:

$$\forall r, d, e \cdot (r, d) \in rd \wedge (r, e) \in re \Rightarrow rd = re$$

The above rule implies that, if the premises are true, then the relationships *rd* and *re* are the same, which also implies that *d* is an object of the same class as *e*. Putting this rule together with axiom 11, we also have that *d* = *e*. The additional meaning is that a viewer object *r* will monitor at most one instance of a viewed object, and not only a single class of the viewed object. These rules characterize the property that we call *viewed singularity*.

²Aggregation relates one container object to a contained one. It is also known as the *has-a* relationship.

Figure 6: A multiple viewers example.

In terms of constraints on the actions of V , the previous property may be stated similarly to the cardinality constraint (i.e., axiom 11). The difference now is that, in the hypothesis – which is stated in the left-hand side of axiom 11, – d_1 and d_2 are not necessarily objects of the same class. For simplicity, d_1 and d_2 are replaced by d and e , which yields:

$$\text{link}(r, d) \wedge \text{link}(r, e) \Rightarrow d = e \quad (14)$$

Alternatively, at the other end of the relationship, a viewed object d may have several viewers attached. There may be not only viewer objects of the same class, but also viewers from different class structures, which implies the existence of different *views-a* relationship theories. This property is called *viewer multiplicity* and is illustrated in Figure 6.

4.5 Viewer vs Viewed Visibility

In the categorical approach described in this chapter, morphisms provide the formal basis to express interconnections between objects in a system, as explained in Section 3.1. Fiadeiro and Maibaum [10] state that two objects of the system interact by sharing some other object, i.e. by having a common sub-component in which they synchronize through morphisms. Similarly, the *views-a* interconnection between viewer and viewed objects is also specified by sharing object signatures. The identification of these shared signatures should be founded on the visibility rules characterizing the *views-a* approach to modeling. These rules determine which part of an object is “visible” or “shared” by both objects.

The signatures in the specification structure of a class may be informally classified according to their specific purposes. Firstly, an object signature involves the attributes and actions which determine the specific object behavior. Actions such as push and pop should be part of the behavior specification of every stack object. Secondly, the signatures are used to model the interconnection activities with other objects.

In a subsystem composed of two object classes interconnected by a *views-a* relationship, the specification structure of a viewed class has only signatures which are relevant to the application being defined. In other words, it has no attribute or action which was specifically intended to access any kind of information maintained by the viewer object. For instance, in a situation where the viewer is an user interface object, the viewed object should not have attributes notifying it about the viewer’s interface-related information. On the other hand, viewer class specifications have not only application-specific signatures (behavior specific), but also have signatures that allow the viewer object to monitor or change the state of a viewed object (interconnection signatures).

We say that each viewer has a sub-object that is identified, or interconnected, with all or part of the viewed object properties to which it is related. As a consequence, this sub-object imposes upon the viewer a pattern of behavior that mimics the behavior of a corresponding viewed object.

Such identification is formalized as shown in the diagram of Figure 5, which describes the combination of relationship (V), viewer (R), and viewed object (D) theories. From this diagram, we observe that the class manager theory M_D synchronizes D and V by means of morphisms. We say that M_D contains a sub-object which is “shared” by both D and V , as it contains a theory that is identified with parts of D and V . On the other side of the relationship, M_R synchronizes V and the viewer class R . Thus, V is synchronized to both the viewer R and the viewed object D . This relationship theory allows the identification of the signatures of related objects and the specification of new axioms based on these signatures.

4.6 Attribute Consistency

Another characteristic of the *views-a* relationship is that it supports the specification of axioms that constrain the values of the attributes in the related objects. This property will guarantee that the states of the objects involved are always consistent among themselves.

Two levels of consistency must be expected when several viewer objects are related to one single viewed object. First, consistency must always exist between each viewer and its related viewed object to assure that the viewer correctly represents the state of a viewed object. A second level of consistency is achieved as a consequence of the first one, that is, all viewers of the same object should be consistent among themselves. Consistency between the viewer object and its related viewed part is called *vertical consistency*, while consistency among the different viewers is called *horizontal consistency*. These consistency properties must be guaranteed by the specification of the *views-a* relationships being defined between the involved objects.

Figure 7 illustrates these consistency properties using a clock application model which contains a counter object with two distinct viewers: a digital clock viewer and an analog clock viewer. In this example, vertical consistency ensures that each viewer object shows the values specified in the attributes of the corresponding viewed object, while horizontal consistency guarantees that the different viewers, i.e., the analog and digital clocks, always show the same time.

4.6.1 Vertical Consistency

We say that two related objects are vertically consistent if their states are coherent with respect to the type of *views-a* relationship established between them. These states are represented by the attribute values of the viewer and viewed objects. For instance, the analog clock viewer of Figure 7 is consistent with the viewed counter only if the attributes associated by means of the *views-a* relationship hold consistent values at any time. In other words, the viewed counter attribute that is mapped by *views-a* to the analog clock viewer should have the same value as the time shown, which is 12:15.

Figure 8 shows some attribute mappings between the Analog Viewer and Viewed Counter objects (see also Figure 7). Such diagram is a simplified instance of the general case shown in Figure 5, where objects and relationship theories are combined into a subsystem theory. In this particular case, the theories of the viewer and viewed objects are interconnected by a *views-a* relationship

Figure 7: *Views-a* Consistencies in a Clock Application.

theory. The class manager theories M_{AR} and M_D are used to synchronize the relationship theory with the viewer and viewed object theories, respectively. Note also that a similar diagram may be obtained for the Digital Viewer and Viewed Counter objects.

Also in this diagram, two attribute morphisms from the class manager theory M_D , which are $t_D \rightarrow time_D$ and $t_D \rightarrow time$, specify the consistency between the viewed object ($time_D$) and the relationship ($time$) attributes. In addition, two attribute morphisms from the class manager theory M_R , which are $t_{AR} \rightarrow time_{AR}$ and $t_{AR} \rightarrow time$, define the consistency between the viewer object and the relationship attributes. These four morphisms guarantee that $time_{AR}$ and $time_D$ will have consistent values at all times.

It is important to mention that consistency between related object states is a constraint property

Figure 8: Vertical consistency through attribute morphisms.

represented by attribute morphisms, as illustrated in Figure 8. These morphisms specify that two attributes of related objects (viewer and viewed) always hold the same value. However, the attribute morphisms do not describe how attribute $time_{AR}$, which is shown in the analog clock viewer, is updated as consequence of a change in the value of attribute $time_D$. In the next subsection, we show how such consistency is achieved by describing how attributes that are modified by actions remain consistent.

4.6.2 Horizontal Consistency

This kind of consistency is among viewers of one viewed object. We say that two viewer objects are horizontally consistent if each of them have attributes that are mapped to one or more common attributes of the same viewed object. In fact, horizontal consistency is a direct consequence of the vertical consistency between each viewer and viewed object.

Still using the application of Figure 7 as example, we see that the analog clock viewer is vertically consistent with the viewed counter state, as the morphisms between the time-related attributes $time_D$ and $time_{AR}$ have shown. For identical reasons, the digital viewer should be vertically consistent with the viewed counter. Consequently, the time-related attributes of both the analog and the digital viewers will be consistent among themselves.

4.7 Action Mappings

In the previous subsection we have used morphisms to obtain attribute (state) consistency between viewer and viewed objects. However, it is forbidden to have only an attribute morphism in a theory presentation, because if we isolate a set of attributes as a sub-object there will be no action to modify their values. This property is a consequence of the locality requirement described in Section 3. As shown in [10], the locality property implies that attributes cannot be separated from the actions that update them, thus imposing a discipline in the way we can interconnect the object theories by means of morphisms.

These conditions imply that the morphism $M_D \Leftrightarrow D$, which was represented in Figure 8, will consist not only of the attribute morphism $t_D \rightarrow time_D$, but also of a set of action morphisms involving all the actions of D which modify the value of the attribute $time_D$. In addition, the specification of the viewer object R should also contain actions that will be identified through morphisms with those actions of D which are synchronized by the morphism $M_D \Leftrightarrow D$. Consequently, when an action act_D of D modifies a given attribute att_D of D , each viewer object R that is monitoring the object D will also execute an action act_R , which is identified with the action act_D , and the corresponding attribute att_R in viewer R will also be modified.

4.7.1 Concurrency Constraints

The formalism we adopt for the specification of the *views-a* modeling approach supports concurrent actions. This concurrency allows us to specify that the viewed and the corresponding viewer attributes are consistent at all times, as we illustrated in Figure 8 using the $time_D$ and $time_{AR}$ attributes. In that case, an action of the object D would modify the attribute $time_D$ at the same time another synchronized action of the object R would update the attribute $time_{AR}$.

Figure 9: A multiple viewers example.

While this kind of action concurrency keeps the attribute values of the objects consistent, conflicting behavior may also result from simultaneous execution of actions. For instance, suppose that while an action of a digital viewer object tries to set the time attribute of the viewed counter, another action of the analog viewer object also tries to set the same time attribute to a different value. This kind of conflict may be resolved by adding some conditions to the relationship theories connecting the different objects.

Our approach to address concurrency conflicts among different viewer objects *viewing* one common part – i.e. a subset of the attributes – of a viewed object is to define the interactions of all the possibly conflicting objects with a viewed object in a single relationship theory. As this relationship theory contains actions which are synchronized with the potentially conflicting actions in related object theories, it is then responsible for defining axioms that constraint the concurrent execution of these conflicting actions. Figure 9 illustrates this approach by interconnecting several viewers R_i with a viewed object D by means of a single *views-a* relationship theory V .

Note that the object model of Figure 6 is very similar to the diagram shown in Figure 9. The difference is that the former had several *views-a* relationship theories, each connecting a pair of objects. Such an approach is suitable when there are no potential conflicts among actions of the different viewers. For example, there may be cases where there are no common attributes in the attribute sets being “monitored” by the distinct viewers. Alternatively, the latter approach uses one single relationship theory for all the objects involved. This approach is generally suitable when the same viewed attributed may be modified by several viewers.

The single relationship and several viewers approach of Figure 9 does not introduce any limitation to the modeling process. All the previously specified properties relating to the viewed class are valid in both modeling approaches.

In the case study presented in Section 5 we exemplify action mappings and the elimination of potential conflicts through axioms constraining the concurrent execution of some viewer actions.

5 Case Study: Dual Interface Clock

In the previous sections we have provided a formal description of the concepts inherent to the *views-a* approach for modeling. While the relationship properties were defined, not much emphasis was given to the actual specification of systems which are based on the *views-a* approach. We specified part of a modeling language, but the actual use of such language was not of immediate concern. Therefore, our current objective is to complement the modeling language definitions with

Figure 10: Interconnection between clock system theories.

the specification of a case study composed of a few objects interconnected by *views-a* relationships.

The case study to be formally specified is the dual interface clock which was used and briefly described in previous sections. This simple system has an application object (Viewed Counter) which is responsible for keeping the correct time of the day, and two different types of user interface for this application. The first interface is an analog time display (Analog Clock Viewer) which has two hands moving on a dial that is divided in twelve sections. This interface does not differentiate the period of the day – i.e. AM or PM, – and with a double mouse click it resets the application counter to 12:00 AM. The second interface is a digital time display that shows the time and period of the day according to the values in the *Counter* object.

Besides the three object class theories, the system formalization involves a few other object theories and morphisms. The system specification, which is presented in the following paragraphs, starts with the viewed object description, and then introduces the two viewer (interface) theories. The relationship and class manager theories, which formalize how objects are put together, are described in sequence. Finally, a few morphisms interconnecting theories are specified. Figure 10 depicts all the theories in the system, which includes the viewer and viewed objects, the class managers, the relationship V , as well as some morphisms. The colimit of this diagram returns a new composite object description, which we call CLOCK-SYSTEM. Note that this diagram is an instance of the general diagram shown in Figure 5, which illustrated the colimit of object and relationship theories.

The specification language structure we will be using in the case study formalization is based on schemas and temporal logic, as described earlier. The first schema is shown in Figure 11. Such schema shows the signatures and axioms of the theory description D of the viewed counter object. Note that the state of D is represented by attributes *time* (which keeps number of hours and minutes in the day) and *period* (which determines whether the time is AM or PM). These attributes are only modified by the actions *set-time*(t), *set-period*(p) and *reset*. The effects of the execution of these actions on the attribute values are shown by the axioms of the theory.

The axioms of a theory may be used to specify constraints, or pre- and post-conditions on the execution of the actions. The first axiom in the specification of theory D specifies an initialization condition on the object, while the following three define post-conditions on the occurrence of the actions, *set-time*(t), *set-period*(p), and *reset*. The last two axioms in D , however, deserve particular attention. For instance, axiom $\neg(\text{set-time}(t) \wedge \text{reset})$ implies that the actions *set-time*(t) and *reset* cannot be executed simultaneously. This mutual exclusion axiom guarantees that these two actions within the two distinct viewers of D must not be executed concurrently if they lead to any kind of inconsistent behavior. Note that within D , the execution of *set-time*(t) modifies the value of the attribute *time* to a value, while the execution of *reset* will modify *time* to a different value. This

Object **D**

Sorts/Functions

TIME : $[1..12] \times [0..59]$;
AM-PM : { AM, PM };

Attributes

time : TIME;
period : AM-PM;

Actions

set-time : TIME;
set-period : AM-PM;
reset;

Axioms

$\boxed{\text{BEG}} \Rightarrow \text{reset};$
 $\text{set-time}(t) \Rightarrow \bigcirc \text{time} = t;$
 $\text{set-period}(p) \Rightarrow \bigcirc \text{period} = p;$
 $\text{reset} \Rightarrow \bigcirc \text{time} = (12, 0) \wedge \bigcirc \text{period} = \text{AM};$
 $\neg (\text{set-time}(t) \wedge \text{reset});$
 $\neg (\text{set-period}(p) \wedge \text{reset});$

End

Figure 11: Specification of the Viewed Counter object.

Object **R_A**

Sorts/Functions

TIME_{AR} : $[1..12] \times [0..59]$;
ANGLE : $[0..359]$;
CHECK-HOURS : TIME_{AR} → ANGLE;
CHECK-MINUTES : TIME_{AR} → ANGLE;

Attributes

time_{AR} : TIME_{AR};
angle_h : ANGLE;
angle_m : ANGLE;

Actions

set-time_{AR} : TIME_{AR};
reset_{AR};
change-angle : TIME_{AR};
double-mouseclick;

Axioms

$\boxed{\text{BEG}} \Rightarrow \text{reset}_{AR};$
 $\text{set-time}_{AR}(t) \Rightarrow \bigcirc \text{time}_{AR} = t;$
 $\text{set-time}_{AR}(t) \Rightarrow \text{change-angle}(t);$
 $\text{reset}_{AR} \Rightarrow \bigcirc \text{time}_{AR} = (12, 0);$
 $\text{reset}_{AR} \Rightarrow \text{change-angle}((12, 0));$
 $\text{change-angle}(t) \Rightarrow \bigcirc \text{angle}_h = \text{CHECK-HOURS}(t);$
 $\text{change-angle}(t) \Rightarrow \bigcirc \text{angle}_m = \text{CHECK-MINUTES}(t);$
 $\text{double-mouseclick} \Rightarrow \text{reset}_{AR};$
 $\neg (\text{set-time}_{AR}(t) \wedge \text{reset}_{AR});$

End

Figure 12: Specification of the Analog Clock Viewer object.

Object R_D

Sorts/Functions

$TIME_{DR} : [1..12] \times [0..59];$
 $AM-PM_{DR} : \{ AM, PM \};$

Attributes

$time_{DR} : TIME_{DR};$
 $period_{DR} : AM-PM_{DR};$
 $time_d : TIME_{DR};$
 $period_d : AM-PM_{DR};$

Actions

$set-time_{DR} : TIME_{DR};$
 $set-period_{DR} : AM-PM_{DR};$
 $reset_{DR};$
 $update-display-time : TIME_{DR};$
 $update-display-period : AM-PM_{DR};$

Axioms

$\boxed{BEG} \Rightarrow reset_{DR};$
 $set-time_{DR}(t) \Rightarrow \bigcirc time_{DR} = t;$
 $set-time_{DR}(t) \Rightarrow update-display-time(t);$
 $set-period_{DR}(p) \Rightarrow \bigcirc period_{DR} = p;$
 $set-period_{DR}(p) \Rightarrow update-display-period(p);$
 $reset_{DR} \Rightarrow \bigcirc time_{DR} = (12, 0) \wedge \bigcirc period_{DR} = AM;$
 $reset_{DR} \Rightarrow update-display-time((12,0)) \wedge update-display-period(AM);$
 $update-display-time(t) \Rightarrow \bigcirc time_d = t;$
 $update-display-period(p) \Rightarrow \bigcirc period_d = p;$
 $\neg (set-time_{DR}(t) \wedge reset_{DR});$
 $\neg (set-period_{DR}(p) \wedge reset_{DR});$

End

Figure 13: Specification of the Digital Clock Viewer object.

potential inconsistent behavior is eliminated with the addition of the mutual exclusion axioms.

A second object specification is shown in Figure 12. In this schema, the analog viewer theory R_A is an interface for the application. Note that part of the structure of R_A , more specifically the signature elements with subscript AR (e.g., $time_{AR}$), is responsible for maintaining the consistency between the states of R_A and D . Such signature elements will be mapped by morphisms which allow the viewer to “observe” the viewed counter object. In addition, the axioms of D involving signatures of the morphism are also preserved in R_A , as this is a requirement of the morphism definition. Some of the system morphisms will be specified later in this section.

The other signature elements of R_A are responsible for user interface activities. The *change-angle* action is called to update the angles of the hands in the analog clock display every time an action changes the attribute $time_{AR}$, which is the only attribute of D being “observed” by the viewer object R_A . This action uses functions $CHECK-HOURS(t)$ and $CHECK-MINUTES(t)$ to calculate the new values of the angle-related attributes. The other action of the interface R_A , which is named *double-mouseclick*, triggers the $reset_{AR}$ action, whenever it is called. As a consequence, $reset_{AR}$ will not only modify the time values in R_A , but it will also trigger the action $reset$ in the object D by means of morphisms. Then, $reset$ will modify the time values of D , thus keeping both viewer and viewed object states consistent.

The third object in the system is the digital viewer object, for which a specification is given in Figure 13. This viewer object is responsible only for monitoring and displaying the time, even though user input events in other interface objects could trigger actions such as $reset_{DR}$ to modify the counter object attribute values. Every time $set-time_{DR}(t)$, $set-period_{DR}(p)$, or $reset_{DR}$ is triggered, the display update changes the time values in accordance with the axioms. The other axioms in the specification are intended to preserve the properties of the viewed object D . Note that R_D , in contrast to R_A , monitors and displays not only the *time* attribute value in D , but also the attribute value of *period*.

In the same way as before (for the Analog Clock Viewer - see Figure 12), part of the structure of R_D , the signature elements with subscript DR (e.g., $time_{DR}$), is responsible for maintaining the consistency between the states of R_D and D . The other signature elements of R_D are responsible for user interface activities. For example, in the previous case (Figure 12) the attribute $time_{AR}$ was used for monitoring and the attributes $angle_h$ and $angle_m$ were used for user interfaces purposes. Now, for the second viewer, we use the attributes $time_{DR}$ and $period_{DR}$ for monitoring and the attributes $time_d$ and $period_d$ for displaying purposes (Figure 13). The actions $update-display-time(t)$ and $update-display-period(p)$ are called to update the time and period values of the digital clock displays $time_d$ and $period_d$.

Having described all the viewed and viewer objects, we now determine the pattern of interaction between these objects. Such pattern, as specified during the software modeling process, should conform to the properties of the *views-a* relationship. This means that all the axioms characterizing a general *views-a* relationship should hold together with additional properties to be specified for this particular case of the clock system.

Figure 14 shows the relationship theory V for our clock system. Parts of the theory defined in Section 4 for the general *views-a* relationship, such as the *link* and *unlink* actions, are now omitted for simplicity. Nevertheless, they are still part of the relationship theory, and so are the *views-a* properties introduced by their related axioms. The purpose of the signatures and axioms shown in the schema V is to synchronize the system objects.

The relationship theory V has two sets of actions: one indexed as $V1$ and the other as $V2$. Both sets act as a cable that connects the viewer objects with the relationship theory. The first cable is connected to the analog viewer theory, while the second one is connected to the digital viewer theory. The distinction between both cables allows the identification of the origin of the triggering of an action and, consequently, the specification of constraints about their execution.

The last axiom in the specification schema V exemplifies a constraint established for the concurrent execution of the viewer actions. Such an axiom states that whenever $set-time_{V1}$, which is connected to $set-time_{AR}$ by morphisms, and $set-time_{V2}$, which is connected to $set-time_{DR}$, occur simultaneously, their parameters must have equal values. This constraint guarantees that no two distinct viewers will concurrently try to set the same counter object to different times, thus generating inconsistent behavior. Note also that there is no concurrency constraint established for $set-period(p)$ as the viewer object R_A does not “monitor” the *period* attribute of D . For a different reason, no concurrency constraint was established for $reset_{V1}$ and $reset_{V2}$, as these actions have no parameters and their concurrent execution generates a consistent modification of the attribute values.

The class manager theories have two distinct purposes. Firstly, all the signatures and axioms

Relationship V

Sorts/Functions

$\text{TIME}_V : [1..12] \times [0..59];$
 $\text{AM-PM}_V : \{ \text{AM}, \text{PM} \};$

Attributes

$\text{time}_V : \text{TIME}_V;$
 $\text{period}_V : \text{AM-PM}_V;$

Actions

$\text{set-time}_{V1} : \text{TIME}_V; \quad \text{set-time}_{V2} : \text{TIME}_V;$
 $\text{reset}_{V1}; \quad \text{set-period}_{V2} : \text{AM-PM}_V;$
 $\text{reset}_{V2};$

Axioms

$\boxed{\text{BEG}} \Rightarrow \text{reset}_V;$
 $\text{set-time}_{V1}(t) \Rightarrow \bigcirc \text{time}_V = t;$
 $\text{set-time}_{V2}(t) \Rightarrow \bigcirc \text{time}_V = t;$
 $\text{set-period}_{V2}(p) \Rightarrow \bigcirc \text{period}_V = p;$
 $\text{reset}_{V1} \Rightarrow \bigcirc \text{time}_V = (12, 0) \wedge \bigcirc \text{period}_V = \text{AM};$
 $\text{reset}_{V2} \Rightarrow \bigcirc \text{time}_V = (12, 0) \wedge \bigcirc \text{period}_V = \text{AM};$
 $\text{set-time}_{V1}(t_1) \wedge \text{set-time}_{V2}(t_2) \rightarrow t_1 = t_2;$

End

Figure 14: Specification of the Views-a relationship.

which controls creation and destruction of all object instances of a class theory. These signatures and axioms were described in Section 3. Secondly, these theories work as synchronization channels between the class theories and the relationship theory V by involving signatures and axioms which are used to maintain consistency between viewers and viewed objects. For example, in the class manager theory M_{AR} which is illustrated in Figure 15, action rst acts as a port that interconnects actions $reset_{AR}$ and $reset_{V1}$ by means of morphisms. Consequently, all actions $reset_{AR}$ of R_A class instances³ are synchronized with both $reset_{V1}$ and the $reset$ action of D .

Note that only the signature of the class manager specification is presented in Figure 15. The axioms for this class manager may be obtained by translations (according to the morphism property preservation requirement) from other object theories in the system, and from axioms defined for class manager theories in Section 3.

There are some morphisms interconnecting the viewer objects R_A and R_D to class manager theories (see Figure 10). The morphism in Figure 16 connects the class manager theory M_{AR} to the viewed object D . There are three morphisms interconnecting class manager theories to the relationship theory V . The first morphism is shown in Figure 17 and connects V with the class manager M_{AR} . This morphism synchronizes the actions of “cable” $V1$ in theory V with actions in M_{AR} . Note that attributes do not need distinct “cables” to be connected, as no additional constraint is required. The second morphism, $M_{DR} \Leftrightarrow V$, connects “cable” $V2$ with theory M_{DR} . The third morphism is $M_D \Leftrightarrow V$. It synchronizes both cables $V1$ and $V2$ to the same actions of D (e.g., $reset_{V1}$ and $reset_{V2}$ are both synchronized to $reset$). There is also one morphism interconnecting a class manager theory (M_D) to the viewed object D (see also Figure 10).

³There is only one instance of R_A in this particular example

Object Signature $\mathbf{M}_{AR}$
Sorts/Functions
 $@R_A;$
 $\mathbf{TIM} : [1..12] \times [0..59];$
Attributes
 $\overline{R}_A : \mathbb{F} @R_A;$
 $\mathbf{tim} : \mathbf{TIM};$
Actions
 $\mathbf{create} : @R_A;$
 $\mathbf{kill} : @R_A;$
 $\mathbf{sttm} : \mathbf{TIM};$
 $\mathbf{rst};$
End

Figure 15: Specification of M_{AR} Class Manager signature.

Morphism $M_{AR} \longrightarrow R_A$
Sorts/Functions
 $\mathbf{TIM} \rightarrow \mathbf{TIME}_{AR};$
Attributes
 $\mathbf{tim} \rightarrow \mathbf{time}_{AR};$
Actions
 $\mathbf{sttm}(t) \rightarrow \mathbf{set-time}_{AR}(t);$
 $\mathbf{rst} \rightarrow \mathbf{reset}_{AR};$
End

Figure 16: A morphism between the Analog Viewer and a Class Manager theory.

Morphism $M_{AR} \longrightarrow V$
Sorts/Functions
 $\mathbf{TIM} \rightarrow \mathbf{TIME}_V;$
Attributes
 $\mathbf{tim} \rightarrow \mathbf{time}_V;$
Actions
 $\mathbf{sttm}(t) \rightarrow \mathbf{set-time}_{V1}(t);$
 $\mathbf{rst} \rightarrow \mathbf{reset}_{V1};$
End

Figure 17: A morphism between *Views-a* and a Class Manager theory.

6 Conclusion

In this paper we have provided a model and properties of a relationship that allows the designer to make explicit the semantics of how different concerns have been separated in an object-oriented design. This model has several advantages. First, the model improves understanding about a reuse mechanism to separate concerns through its formal semantics. Second, it provides some guidance for software designers (through the properties) about how the technique of separating concerns can be applied. Third, the model allows us to reason about separate concerns and their relationships to the system. The reasoning aspects, however, are discussed in another paper [3]. Fourth, the model helps the designer to visualize the assembly operations more clearly. Finally, having a formal model at the design level clarifies how the objects are interconnected rather than having the “glue” buried in a complex programming structure.

Although our goal was not to provide a complete methodology about how to separate concerns in object-oriented design, we believe we have succeeded in exemplifying how a model and properties of a particular reuse technique can improve the understanding and provide more guidance for software designers that want to adopt the technique. The informal counterpart of our model and its associated properties are currently being used in the design of some object-oriented systems, including a map-centered multimedia application [4] and a Web-based education software system [2].

References

- [1] Mehmet Aksit. Separation and Composition of Concerns. *Proceedings of the ACM Workshop on Strategic Directions in Computer Research*, June 1996.
- [2] P.S.C. Alencar, D.D. Cowan, S. Crespo, M.F. Fontoura, and C.J.P. Lucena. A Viewpoint-Based Design Method: A Case Study in Frameworks for Web-based Education. *CS-98, Computer Science Department, University of Waterloo*, 1998.
- [3] P.S.C. Alencar, D.D. Cowan, and C.J.P. Lucena. A Logical Theory of Interfaces and Objects. *revised for IEEE Transactions on Software Engineering*, 1998.
- [4] P.S.C. Alencar, D.D. Cowan, C.J.P. Lucena, and M.A.V. Nelson. An Approach to Hypermap-Based Applications. *Proceedings of the Second International Symposium on Environmental Software Systems (ISESS'97), Whistler, British Columbia*, pages 244–251, 1997.
- [5] J.C. Bicarregui, K.C. Lano, and T.S.E. Maibaum. Associations and Subsystems: A Hierarchical Approach to Encapsulation. *Proceedings of ECOOP'97, Finland*, 1997.
- [6] L. M. F. Carneiro, D. D. Cowan, and C. J. P. Lucena. ADVcharts: a Visual Formalism for Interactive Systems. In *Proceedings of the York Workshop on Formal Methods for Interactive Systems*. Springer, 1994.
- [7] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien. Abstract Data Views. *Structured Programming*, 14(1):1–13, January 1993.

- [8] D. D. Cowan, R. Ierusalimsky, C. J. P. Lucena, and T. M. Stepien. Application Integration: Constructing Composite Applications from Interactive Components. *Software Practice and Experience*, 23(3):255–276, March 1993.
- [9] D. D. Cowan and C. J. P. Lucena. Abstract Data Views: An Interface Specification Concept to Enhance Design. *IEEE Transactions on Software Engineering*, vol. 21, no. 3, pages 229–243, March 1995.
- [10] J. Fiadeiro and T. Maibaum. Temporal Theories as Modularisation Units for Concurrent System Specification. *Formal Aspects of Computing*, 4(3):239–272, 1992.
- [11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [12] W. Hursch and C. V. Lopes. Separation of Concerns. Technical Report NU-CCS-95-3, Computer Science Department, Northeastern University, Boston, MA, 1995.
- [13] Kevin Lano. *Formal Object-Oriented Development*. Springer-Verlag, 1995.
- [14] Wayne C. Lim. Effects of Reuse on Quality, Productivity, and Economics. *IEEE Software*, September 1994.
- [15] Carma McClure. *Software Reuse Techniques*. Prentice Hall, 1997.
- [16] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1992.
- [17] D. J. Reifer. *Practical Software Reuse*. John Wiley and Sons, 1997.
- [18] James Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.